Technology Review: NLTK and Twitter Sentiment Analysis

Vivek Datta (anirban2)
University of Illinois Urbana-Champaign
CS 410: Text Information Systems

Intro:

NLTK (Natural Language Toolkit) is a collection of libraries written in and available to be used with Python for the purpose of conducting several tasks related to NLP (natural language processing). Among these tasks NLTK can be used for, according to the NLTK Project development group, include "classification, tokenization, stemming, tagging, parsing, and semantic reasoning". In this brief paper, we provide a quick introduction of many of the common data preprocessing procedures (tokenization, stemming, and part-of-speech tagging) implemented in NLTK that many developers have come to depend on. In addition, we will focus on a classic problem that happens to be a well-defined use case for NLTK: performing sentiment analysis on Twitter tweets.

Body:

NLTK is commonly used for the purposes of tokenization. Tokenization refers to the act of splitting a particular string of text into its individual substrings, which can consist of words, punctuation marks, and even other sentences themselves, depending on the tokenizer used. These individual components are called tokens, and the library `nltk.tokenize` is able to retrieve them through various different API functions. Some commonly used functions under nltk.tokenize include `word_tokenize` (breaks up an input text into tokens of words), sent_tokenize (breaks up an input text into tokens of sentences), and `wordpunct_tokenize` (similar to `word_tokenizer` except it also converts punctuation into tokens as well). These functions all return a list of tokens, which is amenable to further preprocessing through other NLTK library function calls. For example, if we would like to remove stop words (commonly used words with little informational value), we could retrieve a list of common stopwords from the stopwords module in the `nltk.corpus` library (by calling `stopwords.words('english')`) and subsequently filter out tokens appearing in this stopwords list from the tokens we obtained using `nltk.tokenize`.

NLTK also provides several stemmers that perform stemming, or reducing word families with similar syntactic structure and lexical meanings into their stem, or base, word. For example, the words "dancing" and "danced" can be viably stemmed to the base word "dance". This practice is useful for better query results information retrieval systems and reducing to dimensionality of input data for training machine learning models, all while retaining the word's context. Using the `nltk.stem` library, we are able to import and deploy common stemmers, such as the Porter Stemmer (`PorterStemmer()`), Snowball Stemmer (`SnowballStemmer()`), and the Lancaster Stemmer (`LancasterStemmer()`), for our usage right out of the gate. We can set a variable stemmer to be the result of one of the above stemmer instantiations, and reduce our dictionary of words by iteratively calling `stemmer.stem()` on all available tokens from retrieved from tokenization.

Finally, NLTK can be used for the purposes of part-of-speech (POS) tagging, which is the task of assigning tokens their parts of speech based on their definitions and the surrounding context of how the word token is used within the sentence it belongs to. POS tagging is especially useful for a variety of NLP applications, as it allows one to glean both syntactic and semantic insights of specific words and use this information to eventually generate parse trees of sentences. From the main `nltk` library, one can import the `pos_tag` function, which performs this task for us. The `pos_tag` function takes in a list of tokens and returns a list of tuples, where each tuple is a pair of both a token and its corresponding part-of-speech that the token has been assigned.

With these insights on how to leverage NLTK for common data preprocessing taks, we can now turn our attention towards using this toolkit for sentiment analysis, or determining the polarity of given body of text and whether it expresses a positive or negative views or attitudes. Without diving into the full technical implementation, we will use NLKT and Scikit-learn to give a high-level overview on how to build a binary classifier that characterizes a given Twitter tweet as expressing either a positive or negative sentiment.

First, we need to conduct our standard data preprocessing tasks before building our model, which is where we are able to use the NLTK functions we have covered earlier. After loading up our training corpus of Twitter tweets and their corresponding sentiment labels as a data frame, we can conduct the preprocessing workflow of tokenizing each tweet using `word_tokenize`, removing all of the stop words by checking which tokens also have membership in `stopwords.words('english')`, and normalizing each word to its stem by using any stemmer available under the `nltk.stem` package. Recall that tokenization converts a string to a list of tokens, so we must retrieve the string of each tweet preprocessed using NLTK by calling `''.join(preprocessed_tokens)` on the list of our preprocessed tweet tokens.

From here, we are finally able to vectorize our data and feed both this numeric representation of our Twitter tweets into our binary classifier model as part of training. Using Scikit-learn, we can use the `TfidfVectorizer` module, which sequentially converts our corpus of tweets into a matrix of counts for each word in our dictionary (`CountVectorizer`) followed by reweighting the counts using the TF-IDF heuristics (`TfidfTransformer`). Then, we can use a classification model of our choice, such as `MultinomialNB` or `LogisticRegression`, to eventually train the model, using both our input matrix formed by vectorizing our Twitter data corpus preprocessed using `NLTK` as well as the corresponding sentiment labels as our training target output. Finally, we can perform these same steps of data preprocessing and TF-IDF vectorization on a corpus of held-out Twitter tweets, and can use the model we have just trained to ultimately obtain a set of predictions regarding the polarity and sentiment of each tweet.

Conclusion:

In this review, we have discussed how to utilize NLTK for a wide variety of common data preprocessing tasks necessary for many NLP problems. Specifically, we discussed how NLTK can be used for tokeniziation, removing stopwords, stemming tokens, and assigning these lexical units the parts of speech they are likely to belong to. We then discussed the convenience and practicality of NLTK function calls (combined with models and vectorizers available via Scikit-learn) when performing sentiment analysis on Twitter tweets, giving a sample workflow of how to easily preprocess an input corpus of tweets before eventually vectorizing and training a classification model of the tweets' polarity/sentiment. From this, one can reasonably conclude that NLTK is a platform that increases the ease of converting text to machine-interpretable representations leveragable by downstream models, helping to explain its wide adoption from academic research to industry-level contexts.

Citations:

- https://towardsdatascience.com/intro-to-nltk-for-nlp-with-python-87da6670dde
- https://www.nltk.org/api/nltk.tokenize.html
- https://www.nltk.org/howto/stem.html
- https://www.nltk.org/book/ch05.html
- https://medium.com/swlh/tweet-sentiment-analysis-using-python-for-complete-beginners-4aeb4456040
- https://towardsdatascience.com/twitter-sentiment-analysis-classification-using-nltk-python-fa912578614c
- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer